Git

De Wikipedia, la enciclopedia libre

Git (pronunciado "guit"[2]) es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente. Al principio, Git se pensó como un motor de bajo nivel sobre el cual otros pudieran escribir la interfaz de usuario o front end como Cogito o StGIT. [3] Sin embargo, Git se ha convertido desde entonces en un sistema de control de versiones con funcionalidad plena. [4] Hay algunos proyectos de mucha relevancia que ya usan Git, en particular, el grupo de programación del núcleo Linux.

El mantenimiento del software Git está actualmente (2009) supervisado por Junio Hamano, quien recibe contribuciones al código de alrededor de 280 programadores.

Git

Desarrollador

Junio Hamano, Linus Torvalds

http://git-scm.com/

Información general

Diseñador	Linus Torvalds
Última versión estable	2.6.1 (info (http://git-scm.com/)) 5 de octubre de 2015
Género	Control de versiones
Programado en	C, Bourne Shell, Perl[1]
Licencia	GNU GPL v2

[editar datos en Wikidata]

Índice

Características

El diseño de Git se basó en BitKeeper y en Monotone. [5] [6]

El diseño de Git resulta de la experiencia del diseñador de Linux, Linus Torvalds, manteniendo una enorme cantidad de código distribuida y gestionada por mucha gente, que incide en numerosos detalles de rendimiento, y de la necesidad de rapidez en una primera implementación.

Entre las características más relevantes se encuentran:

- Fuerte apoyo al desarrollo no lineal, por ende rapidez en la gestión de ramas y mezclado de diferentes versiones. Git incluye herramientas específicas para navegar y visualizar un historial de desarrollo no lineal. Una presunción fundamental en Git es que un cambio será fusionado mucho más frecuentemente de lo que se escribe originalmente, conforme se pasa entre varios programadores que lo revisan.
- Gestión distribuida. Al igual que Darcs, BitKeeper, Mercurial, SVK, Bazaar y Monotone, Git le da a cada programador una copia local del historial del desarrollo entero, y los cambios se propagan entre los repositorios locales. Los cambios se importan como ramas adicionales y pueden ser fusionados en

la misma manera que se hace con la rama local.
- Los almacenes de información pueden publicarse por HTTP, FTP, rsync o mediante un protocolo nativo, ya sea a través de una conexión TCP/IP simple o a través de cifrado SSH. Git también puede emular servidores CVS, lo que habilita el uso de clientes CVS pre-existentes y módulos IDE para CVS pre-existentes en el acceso de repositorios Git.
- Los repositorios Subversion y svk se pueden usar directamente con git-svn.
- Gestión eficiente de proyectos grandes, dada la rapidez de gestión de diferencias entre archivos, entre otras mejoras de optimización de velocidad de ejecución.
- Todas las versiones previas a un cambio determinado, implican la notificación de un cambio posterior en cualquiera de ellas a ese cambio (denominado autenticación criptográfica de historial). Esto existía en Monotone.
- Resulta algo más caro trabajar con ficheros concretos frente a proyectos, eso diferencia el trabajo frente a CVS, que trabaja con base en cambios de fichero, pero mejora el trabajo con afectaciones de código que concurren en operaciones similares en varios archivos.
- Los renombrados se trabajan basándose en similitudes entre ficheros, aparte de nombres de ficheros, pero no se hacen marcas explícitas de cambios de nombre con base en supuestos nombres únicos de nodos de sistema de ficheros, lo que evita posibles, y posiblemente desastrosas, coincidencias de ficheros diferentes en un único nombre.
- Realmacenamiento periódico en paquetes (ficheros). Esto es relativamente eficiente para escritura de cambios y relativamente ineficiente para lectura si el reempaquetado (con base en diferencias) no ocurre cada cierto tiempo.

Órdenes básicas

- `git fetch`:

Descarga los cambios realizados en el repositorio remoto.

- `git merge <nombre_rama>`:

Impacta en la rama en la que te encuentras parado, los cambios realizados en la rama “nombre_rama”.

- `git pull`:

Unifica los comandos *fetch* y *merge* en un único comando.

- `git commit -am "<mensaje>"`:

Confirma los cambios realizados. El “mensaje” generalmente se usa para asociar al *commit* una breve descripción de los cambios realizados.

- `git push origin <nombre_rama>`:

Sube la rama “nombre_rama” al servidor remoto.

- `git status`:

Muestra el estado actual de la rama, como los cambios que hay sin commitear.

- `git add <nombre_archivo>`:

Comienza a trackear el archivo “nombre_archivo”.

- `git checkout -b <nombre_rama_nueva>`:

Crea una rama a partir de la que te encuentres parado con el nombre “nombre_rama_nueva”, y luego salta sobre la rama nueva, por lo que quedas parado en ésta última.

- `git checkout -t origin/<nombre_rama>`:

Si existe una rama remota de nombre “nombre_rama”, al ejecutar este comando se crea una rama local con el nombre “nombre_rama” para hacer un seguimiento de la rama remota con el mismo nombre.

- `git branch`:

Lista todas las ramas locales.

- `git branch -a`:

Lista todas las ramas locales y remotas.

- `git branch -d <nombre_rama>`:

Elimina la rama local con el nombre “nombre_rama”.

- `git push origin :<nombre_rama>`:

Elimina la rama remote con el nombre “nombre_rama”.

- `git remote prune origin`:

Actualiza tu repositorio remoto en caso que algún otro desarrollador haya eliminado alguna rama remota.

- `git reset --hard HEAD`:

Elimina los cambios realizados que aún no se hayan hecho *commit*.

- `git revert <hash_commit>`:

Revierte el *commit* realizado, identificado por el “hash_commit”.

Buenas prácticas

Cada desarrollador o equipo de desarrollo puede hacer uso de Git de la forma que le parezca conveniente. Sin embargo una buena práctica es la siguiente:

Se deben utilizar 4 tipos de ramas: Master, Development, Features, y Hotfix.

- Master:

Es la rama principal. Contiene el repositorio que se encuentra publicado en producción, por lo que debe estar siempre estable.

- Development:

Es una rama sacada de master. Es la rama de integración, todas las nuevas funcionalidades se deben integrar en esta rama. Luego que se realice la integración y se corrijan los errores (en caso de haber alguno), es decir que la rama se encuentre estable, se puede hacer un merge de development sobre la rama master.

- Features:

Cada nueva funcionalidad se debe realizar en una rama nueva, específica para esa funcionalidad. Estas se deben sacar de development. Una vez que la funcionalidad esté pronta, se hace un merge de la rama sobre development, donde se integrará con las demás funcionalidades.

- Hotfix:

Son bugs que surgen en producción, por lo que se deben arreglar y publicar de forma urgente. Es por ello, que son ramas sacadas de master. Una vez corregido el error, se debe hacer un merge de la rama sobre master. Al final, para que no quede desactualizada, se debe realizar el merge de master sobre development.

Véase también

- Control de versiones
- Programas para control de versiones

Referencias

1. «git/git.git/tree» (http://git.kernel.org/?p=git/git.git;a=tree). git.kernel.org. Consultado el 15 de junio de 2009.
2. «Tech Talk: Linus Torvalds on git (at 00:01:30)» (https://www.youtube.com/watch?v=4XpnKHJAok8&t=1m30s). YouTube. Consultado el 20 de julio de 2014.
3. Linus Torvalds (8 de abril de 2005). «Re: Kernel SCM saga» (http://marc.info/?l=linux-kernel&m=111293537202443).
4. Linus Torvalds (23 de marzo de 2006). «Re: Errors GITtifying GCC and Binutils» (http://marc.info/?l=git&m=114314642000462).
5. Linus Torvalds (5 de mayo de 2006). «Re: [ANNOUNCE] Git wiki» (http://marc.info/?l=git&m=114685143200012). Referencias de los antecesores de Git
6. LKML: Linus Torvalds: Re: Kernel SCM saga (http://lkml.org/lkml/2005/4/8/9)

Enlaces externos

- Sitio web oficial (http://git-scm.com/)